format and timing. Most of the effort went into reading and writing object files produced by assemblers/compilers in various formats, providing capability for editing data, and creating a convenient user interface. The resulting program, `PICPGM84.EXE`, achieves these results. A basic version of this program is available for downloading from the Circuit Cellar BBS. A registered version with more features, including support for the PIC16C71, is available on disk.

The "simple" part of the software is talking to the SPA (or, equivalently, to the on-board programming circuitry, since they look identical to the software). Since the rate of the PIC16C84's programming clock permits operation at DC to 5 MHz, there is nothing critical about the "bit-banging" used to generate the data and clock signals fed to RB.7 and RB.6. Required setup and hold times of 100 ns are easily achieved through normal program delays, even on the fastest of PCs. The required intercommand gap of 1 µs minimum is also achieved through program delays.

As I mentioned previously, there is no need for special efforts to push the clock rate. As implemented, the time needed to send a typical command and its data is around 1 ms. This is small compared to the 10 ms per byte required by the EEPROM's internal timer. Be sure to note that the 10-ms programming period is a target value, but early chips did not meet this spec. To allow for slower (and future, faster) devices, the software permits the user to specify any duration in increments of 1 ms. A side benefit of slower clock rates is that cable length becomes much less critical, permitting more convenient location of the adapter. Use of Schmitt-trigger inputs by the PIC16C84 on RB.7 and RB.6 in programming mode ensures clean operation.

Since the PIC16C84 in serial program mode does not possess a "bulk erase" capability (although it *is* available in parallel mode), we normally program the full 1K program memory to ensure that any old data is completely overwritten. This operation takes typically less than 15

The Data Format Quandary

Intel hex formats have been used for years as the standard for object files sent to device programmers. Probably the most popular one is INHX8M (INtel HeX 8-bit Merged). This is a byte-oriented protocol and requires special definition to support the 14-bit PIC program data. All characters in the file are ASCII characters, which makes it simple to inspect and process the data. The format is:

:NNAAAATTXXXX..XXCC

Here, ":" is a lead-in character which says that a block of data follows. *NN* is a byte count, *AAAA* is a starting address for the data in the block, *TT* is a two hex-digit (ASCII) field which specifies the type of data. It is always 00 except for the end-of-file terminator where it is 01. Each pair of ASCII characters *XX* specifies one data byte, and *CC* is a one-byte checksum. A typical block might look like:

:10123400001122334455667788 99AABBCCDDEEFFB2

This line specifies that 16 (decimal) bytes are contained in the block. The starting address for data is 1234h. The next 00 is the flag that says this is normal data. The following 32 ASCII characters specify 16 bytes of data (00h to FFh). The last pair of characters is a checksum.

Even though the characters in the file are ASCII, the checksum is performed on the hex digits they represent. The sum of all bytes (including the checksum, but not including the ":") must add up to zero. The checksum is chosen to make this happen. In the above example, the sum of all the bytes preceding the checksum is 84E (hex). We drop all but the two least-significant digits, which gives 4E. The checksum must then be B2, since when we add this to 4E we get 00 (ignoring the carry).

A typical terminator line looks like:

:00000001FF

This signifies that we have no data bytes, a dummy address of 0000, the terminator flag 01, and the checksum FF. The special block produced by the Parallax PASM71 assembler for ID and configuration information might look like:

:050FFB004142434413D4

This indicates five bytes of data at a starting location of 0FFB. The data bytes 41–44 represent ASCII characters "A" through "D" for the ID code. The fifth byte, 13, is configuration bit data. The checksum is D4. Only a program which recognizes that address 0FFB represents configuration information (and that ID bytes occupy only the low seven bits with the high seven filled with ones) would be able to make proper use of this block. Certainly not a very universal approach to life.

Note that INHX8M is a byte-oriented format. Addresses are byte addresses and data are individual bytes. How do these data relate to the 14-bit word of the PIC16C84 (and others)? The 14-bit word is assumed to be right-justified in a 16-bit word. This word is split into two bytes with the low-order byte coming first in the line of data followed by the high-order byte. They thus appear swapped. The four-digit addresses, however, do *not* have their bytes swapped. Note that these are byte addresses and, hence, are double the word addresses of the program.

seconds. For special situations in which only a portion of the EEPROM needs to be modified, the programming software permits the user to specify a partial region. Additionally, any region of the user EEPROM area as well as the ID and configuration bits may be read or programmed. Data read from the PIC16C84 may be saved as a hex file.

One thing I soon learned is that there is not complete standardization with respect to object (hex) file structure. While Microchip often specifies INHX8M (INtel HeX 8-bit Merged) format for data (see the sidebar), 16-bit formats and even binary files are sometimes employed. This makes some sense since the PIC16C84 actually uses 14-bit data words. However, rather than complicate the programming software unduly and restrict ultimate compatibility, only the popular INHX8M format is used by PICPGM84. Conversion programs are available or may be written to translate other formats into this standard.

Another area of incompatibility that arose during development is in specifying user-data EEPROM, ID codes, and configuration bit data. Microchip encourages this information to be included in object files, but the actual means is not totally standardized. User-data EEPROM is a byte-wide area, thus INHX8M is a natural format to use for it. ID codes are actually 14 bits in size, although they recommend using only the low-order 7 bits so ID information may be verified even if the code protection bit is programmed. The configuration bit word is 14 bits wide, but only the least-significant 5 bits are used. As an example of the incompatibility that exists, the Parallax PASM71 assembler, for example, produces a special 5-byte block in the hex file for ID and configuration bit data, and assigns it to address 0FFBh, as shown in the following sample block:

```
:050FFB004142434411D6
ID = "ABCD"  FUSES = 11h
```

This approach restricts ID codes to single bytes.

Intel 16-bit hex format (INHX16) is similar, but the high and the low data bytes are not swapped nor are the addresses doubled. This format would seem to be more natural for the 14-bit PIC. However, recall that the user data EEPROM area is only byte wide.

Binary format is a file (usually with a .BIN extension) which contains pure 16-bit binary numbers with no markers, addresses, or checksums. It represents a block of data which could reside anywhere in memory. Thus, we need to know the starting location, which for most purposes is usually taken to be 0000. For large amounts of data, the file is more compact, but it lacks useful information for a programmer. It also implies that an entire memory map is supplied, which can be wasteful if program data, user memory, and configuration information are mapped to disjoint areas as with the PIC. Furthermore, not being an ASCII file, it is difficult to read or edit using simple tools.

The most straightforward approach to follow is the address mapping suggested by Microchip. This places program data at 0000h–03FFh, user data at 2100h–2140h, ID codes at 2000h–2003h, and configuration bit data at 2007h. If the object file follows these conventions, PICPGM84 will properly read all this information automatically. If not specified, PICPGM84 sets all unused program memory to 3FFFh, data memory to FFh, and ID codes to 007Fh, just as a blank chip would look. It leaves all configuration bits unprogrammed (1Fh). This configures the PIC16C84 for no code protection, power-up and watchdog timers enabled, and RC-type oscillator.

If the assembler or compiler produces object code for these special areas using a different convention (as I found with the Parallax PASM71 assembler), either the object file must be passed through a translator to produce a file meeting standard convention, or the data may be patched by hand using PICPGM84's editing capabilities.

PICPGM84 also permits the reading of just the user-data EEPROM and configuration bit data from a file. This capability is useful in at least two situations. At times, the user data and bit data may be fine and only the program needs changing. Other times, the program is cast in concrete and only the user data needs reprogramming, for example to load calibration data or serial numbers. Combined with the partial programming capability mentioned earlier, this offers a lot of flexibility for the user.

I wrote the PICPGM84 software in C to run on any IBM-compatible PC regardless of its speed. It calibrates delay loops based on the CPU's speed. When programming PIC16C84s (with their internally timed EEPROM program cycle), operation under Windows or OS/2 with other activities generating interrupts should be possible. But if used to program the PIC16C71 (with its tight program pulse requirements), it should run as a stand-alone program with TSRs and other devices inactive. In all cases, no other devices should attempt to use the LPT1 port whenever the SPA is attached. Otherwise, inadvertent programming could take place.

The program occupies around 60K and requires EGA or better video operating in 640×200×16-color mode. As with most software, PICPGM84 is an evolving program, and if there is interest, I will make more advanced versions available in the future.

CONCLUSION

With the PIC16C84 Serial Programming Adapter (PICSPA84), Prototyping Board (PICPRO84), and PICPGM84 software, getting started in the fun of applying PICs to embedded applications just got a whole lot simpler and less expensive. No more expensive windowed/erasable chips, no more UV erasers, no more programmers costing over $100, and no more removing chips from the board for reprogramming.